Update propagation strategies for high-performance OLTP

Caetano Sauer[1], Lucas Lersch[2]*, Theo Härder[1], and Goetz Graefe[3]

[1] TU Kaiserslautern, Germany {csauer,haerder}@cs.uni-kl.de
[2] TU Dresden & SAP AG lucas.lersch@sap.com
[3] Hewlett Packard Laboratories goetz.graefe@hpe.com

Abstract. Traditional transaction processing architectures employ a buffer pool where page updates are absorbed in main memory and asynchronously propagated to the persistent database. In a scenario where transaction throughput is limited by I/O bandwidth—which was typical when OLTP systems first arrived—such propagation usually happens on demand, as a consequence of evicting a page. However, as the cost of main memory decreases and larger portions of an application's working set fit into the buffer pool, running transactions are less likely to depend on page I/O to make progress. In this scenario, update propagation plays a more independent and proactive role, where the main goal is to control the amount of cached dirty data. This is crucial to maintain high performance as well as to reduce recovery time in case of a system failure. In this paper, we analyze different propagation strategies and measure their effectiveness in reducing the number of dirty pages in the buffer pool. We show that typical strategies have a complex parametrization space, yet fail to robustly deliver high propagation rates. As a solution, we propose a propagation strategy based on efficient log replay rather than writing page images from the buffer pool. This novel technique not only maximizes propagation efficiency, but also has interesting properties that can be exploited for novel logging and recovery schemes.

1 Introduction

Database systems rely on persistent storage to provide the durability property of "ACID" transactions. However, in order to deliver acceptable performance, operations that modify data are usually performed in a volatile copy of data objects in the buffer pool and later propagated to persistent storage. In a *force* approach [5], such propagation happens at commit time at the latest, whereas a *no-force* approach—which is used in the vast majority of database systems—delays such propagation to an arbitrary point in time, relying on REDO logging to provide durability. In the latter case, which is the focus of this paper, controlling this delay is crucial for two main reasons: (1) it enables the efficient recycling of buffer pool frames and log space for new transactions; and (2) it determines the amount of recovery effort in case of a system failure.

* Work done while at TU Kaiserlsautern

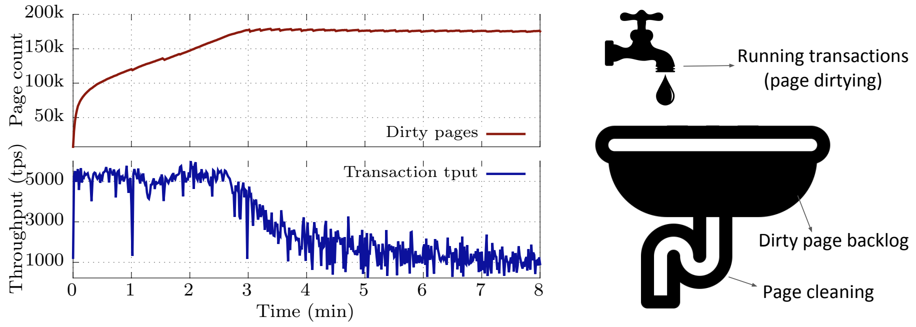

Fig. 1: Dirty page backlog and its implication on system performance

Traditionally, main memory has been a limited resource, so that transaction throughput was limited by the bandwidth of page read and write operations. In this scenario, update propagation is almost exclusively used for reason 1 above: cached pages must be evicted from the buffer pool to make room for pages accessed by new transactions and, as a consequence, its updates are propagated to persistent storage. However, as the capacity of main memory increases, which has been a strong trend in the past years, typical workloads are less likely to depend on page I/O to make progress. In this scenario, reason 1 becomes less important, and the main role of update propagation becomes the minimization of recovery effort—or reason 2 above. This work is motivated by the need to re-evaluate update propagation strategies for this new crucial role.

The problem addressed in this work can be characterized by a race between user transactions that modify pages and mark them dirty and system actions that clean these pages. If the *cleaning speed*, i.e., the number of pages being cleaned per second, does not match the *dirtying speed* of the workload, the accumulated backlog may negatively impact system performance. This backlog can be measured across two dimensions: number of dirty pages and accumulated log volume. In the latter case, the issue appears when the log volume required for REDO recovery fills up the entire log device, so that transactions cannot make progress until some log space is freed. Since the length of REDO recovery is determined by the dirty pages in the buffer pool, this means that an inefficiency in page cleaning can lead to a complete halt of read-write transactions. In the former case, if the number of dirty pages grows until it fills up the entire buffer pool, the system eventually slows down as transactions must wait for page eviction, despite their working set fitting into main memory. This can happen, for instance, in the TPC-C workload, whose working set consists mainly of warehouse and customer data as well as currently active orders. If page cleaning is inefficient, dirty pages containing finished orders will linger in the buffer pool, until no clean frames are available for inserting new orders and the system slows down, becoming I/O-constrained even though there is abundant main memory to hold the working set.

Figure 1 presents the problem graphically in two ways. On the right-hand side, the problem is illustrated as an analogy of a sink full of water—running transactions that make clean pages dirty are like a faucet filling up the sink, while page cleaning corresponds to the drain. If the drain is not large enough, water will accumulate in the sink, which in our case corresponds to the backlog discussed above. Eventually, the sink fills up and the only way to avoid an overflow is to close the faucet, i.e., the transaction throughput must be reduced. On the left-hand side, the problem is shown in a real experiment which plots both the number of dirty pages in the buffer pool as well as the transaction throughput over time. On the top graph, the number of dirty pages grows until it reaches the buffer pool size of 180,000 pages. At that point, which occurs at minute 3 of the experiment, the transaction throughput drops substantially, from 5,000 to about 1,000 transactions per second. Such drop in throughput is a direct consequence of the page cleaner not being able keep up with the running transactions.

This work makes two main contributions. First, we discuss and evaluate typical propagation strategies that write pages from the buffer pool into persistent storage—this is the common technique used in state-of-the-art database systems, and we refer to it as *page-based propagation*. Second, we propose a novel technique which propagates updates by replaying REDO log records in an efficient way—we call this *log-based propagation*. The key to enabling this new technique is a *partially sorted log*, which was introduced in previous work in the context of archiving and recovery from media failures [11]. Rather than employing the partially sorted organization only for media recovery, we exploit its log replay efficiency to propagate updates as well, achieving a propagation strategy which is completely decoupled from the buffer pool. An empirical evaluation of the new method shows that it performs better than traditional strategies, maintaining a controlled dirty page backlog.

In the remainder of this paper, Section 2 summarizes related work, including a brief discussion of background techniques on which our approach is based, a related family of instant recovery algorithms, and alternative approaches for in-memory database system designs. Section 3 discusses page-based propagation strategies, while Section 4 introduces our novel log-based approach. Experiments that support our claims empirically are provided in Section 5. Finally, Section 6 discusses future work opportunities and concludes our findings.

2 Related work

We divide related work into three main categories. First, we discuss the basic system architecture on which our approach is based. Second, we briefly summarize a family of techniques known as instant recovery [3]. Our approach for log-based propagation relies on a log organization proposed for one of such techniques. Third, we summarize update propagation strategies as implemented in main-memory database system designs found in the literature.

2.1 Background

Our approach is based on a traditional database system architecture, with page-based data structures accessed via a buffer pool backed by SSD or HDD devices [6]. Write-ahead logging with physiological log records as implemented in ARIES [9] is also assumed. Since this work concerns only buffer management and storage, it is orthogonal to concurrency control schemes—for both transaction isolation and multi-threaded data structure access.

We assume that a system thread called *page cleaner* is responsible for flushing pages from the buffer pool. Multiple threads can be used for multiple storage drives. Checkpoints are of the fuzzy type and do not flush dirty pages [10]. If page replacement is required, user threads simply wake up the cleaning service and wait for a signal of completion. This design allows a centralization of all cleaning aspects to a single system module. The page cleaner generates log records for each write operation, which allows a more precise computation of the dirty page set during checkpoints and log analysis, thus reducing the recovery effort in case of a system failure [10].

2.2 Instant recovery techniques

A family of techniques known as *instant recovery* enables incremental, on-demand recovery of individual pages from both system and media failures [3]. Our approach for log-based propagation is based on the partially sorted log data structure, as employed in single-pass restore for the log archive [11]. However, it goes beyond the scope of media recovery, relying on the partially sorted log for update propagation during normal processing. As such, the partially sorted log should be kept on lower-latency devices such as SSDs instead of on archive storage. This also allows its usage for restart after a system failure and single-page repair [3], since it provides faster log replay in general.

A further instant recovery technique known as *write elision* permits the eviction of dirty pages from the buffer pool without flushing them first [3]. This leaves the persistent page image out of date, and requires single-page repair the next time it is fetched. In principle, write elision alleviates the backlog problem introduced in Figure 1, because running transactions need not wait for a page flush before acquiring an empty buffer pool frame. However, since the page on disk remains out of date, its old log records cannot be recycled until the page is repaired. This means that write elision reduces the dirty page backlog but not the log backlog, and the situation depicted in Figure 1 is likely to happen anyway, unless the system has a chance to catch up during lower activity periods.

Rather than being an alternative technique, write elision complements log-based propagation in which it eliminates the need to ever flush a page from the buffer pool. Furthermore, it permits fast reaction in situations of memory pressure, where evicting dirty pages is a better choice than evicting clean but frequently accessed ones. Further advantages of combining log-based propagation and write elision are discussed in Section 6 as future work.

2.3 In-memory database systems

In-memory database systems are built on the assumption that the entire dataset fits into main memory, but persistent storage is still required to provide transaction durability. As such, some form of update propagation is still required, and the backlog problem still exists in some form or another. Early work by Levy and Silberschatz [7] already recognized the problem of page-based propagation schemes in the context of main-memory databases. They proposed a log-based approach similar to the one introduced in this work, but because the log is not sorted or prepared in any way, log replay requires random I/O operations, which can be multiple orders of magnitude slower than the page dirtying rate in main memory. To circumvent this problem, the authors suggest increasing I/O bandwidth with multiple disks in a striped configuration, but not only is the required amount of disks impractical, it would be very sensitive to skew, thus not distributing the I/O operations equally among disks. Our log-based propagation approach fully utilizes the sequential write speed of a single device, thus being more efficient and feasible.

The traditional propagation approach in most main-memory DBMS designs is to maintain action- or transaction-consistent checkpoints [5] on persistent storage. Propagation to this checkpoint should be performed concurrently to transaction activity. A common approach—which is present in both early [1] and modern [8] designs—is to put the database in a temporary copy-on-write mode, flushing shadow versions of pages to the checkpoint file while transactions make updates on copied images. The problem addressed in this research is thus also present in such systems, since checkpointing of in-memory databases is very similar to page cleaning as discussed here—the end goal is always to increase propagation efficiency and diminish recovery times in case of failure.

As observed in recent research [4], the assumption of all data fitting in main memory is unrealistic, and techniques of traditional disk-based systems—when adapted for better in-memory performance—may be a better alternative to techniques of main-memory DBMSs. This is especially true for recovery, since many such systems have very inefficient and incomplete (in the sense that media failures are not considered) recovery schemes. This research represents a step in the direction of optimizing traditional techniques for large memories, while still supporting disk-resident data with high reliability.

3 Page-based propagation strategies

As discussed in Section 2.1, page-based propagation is performed by the page cleaner service. This section provides an overview of how the page cleaner works, including its impact on the recovery effort in case of a system failure. Furthermore, we discuss a variety of policies that can be implemented to achieve the two, sometimes conflicting, goals of page cleaning: reducing dirty page backlog and recovery effort.

3.1 Page cleaner algorithm

The page cleaner is an independent system thread, which runs continually in a main loop described in Algorithm 1. First, it waits for an activation signal, which may come from threads waiting for eviction or log space recycling, or a timeout if it is set to run periodically. Once activated, the cleaner collects a list of candidate frame descriptors in a priority queue. This queue is used to order frames according to some policy, such as oldest-first or hottest-first; these are discussed in detail in Section 3.2.

Algorithm 1 Page cleaner main loop

```
 1: procedure PageCleaner(bufferPool, policy, maxCandidates)
 2:     waitForActivation()
 3:     candidates ← createHeap(policy, maxCandidates)
 4:     writeBuffer ← allocateBuffer()
 5:     for all d in bufferPool.descriptors do
 6:         if d.isDirty() then
 7:             candidates.pushHeap(d)
 8:         end if
 9:     end for
10:     clusters ← sortAndAggregateByPageID(candidates)
11:     cleanLSN ← logTailLSN()
12:     for all c in clusters do
13:         latchAndCopy(c, writeBuffer)
14:         flush(writeBuffer)
15:         logPageFlush(c, cleanLSN)
16:         bufferPool.updateCleanLSN(c, cleanLSN)
17:     end for
18: end procedure
```

Once a list of candidates is collected, it is sorted by page ID in line 10 of Algorithm 1. The purpose here is to form clusters of adjacent pages, which can be flushed with a single write operation. For each cluster of pages, the cleaner then latches their buffer pool frames in shared mode and copies their contents into its internal write buffer. This is done to avoid holding a latch, and thus delaying updating threads, for the entire duration of a synchronous write, which is performed in line 14. These writes must be synchronous because marking a page as clean before it is actually persisted may result in lost updates in case of a system failure. After the flush operation completes, it is logged to support a more precise estimation of the dirty page set during log analysis [10]. This step is not required, but has benefits for more efficient recovery.

The last step of the algorithm is to mark the page as clean in the buffer pool. Traditionally, the dirty state of each page is tracked with a Boolean flag on each frame descriptor. Before setting the dirty flag to $false$, the cleaner must check whether or not an update happened to the page while it was being flushed. An alternative approach, which is used in our design, is to maintain an additional LSN field instead of a Boolean flag in the page descriptor. This field, called *CleanLSN*, contains some LSN value for which all previous updates on the page are guaranteed to have been propagated; it is initialized with the *PageLSN* value and updated by the cleaner every time a page is flushed. Using this mechanism, a page is considered dirty if and only if $PageLSN > CleanLSN$. This approach eliminates the need to keep track of PageLSN values of copied page images, and can also be used to implement a cleaning policy that considers "how long ago" a page was last flushed.

3.2 Page cleaning policies

Before discussing different page cleaning policies—and why it is important to have them instead of collecting all dirty pages as candidates—it is important to understand the impact that the cleaner has on the dirty page backlog and, ultimately, on the recovery effort in case of a system crash.

The main efficiency measure of the page cleaner is its write bandwidth, i.e., how many pages it can write per second (or how large the "drain" is in the sink analogy of Figure 1). However, optimizing for write bandwidth does not necessarily minimize the dirty page backlog, because—as mentioned in Section 1—the backlog can be measured as not only the number of dirty pages, but also how much log volume is covered by such pages. If the cleaner policy in use neglects the log volume, the length of the REDO log scan required during recovery is not kept under control, and a situation similar to that of Figure 1 may happen when the log device is full. Therefore, the goal of page cleaning policies is to reduce the dirty page backlog—and consequently reducing the recovery effort—across two dimensions: number of dirty pages and log volume.

A page cleaning policy can be defined as a sort order applied to candidate buffer pool frames. This is implemented using a priority queue in the $pushHeap$ function of Algorithm 1. Our work considers three policies: oldest first (lowest CleanLSN value); coldest first (lowest reference counter value); and hottest first (highest reference counter value). Each of these policies has its own benefits for reducing the dirty page backlog. An empirical analysis is performed in Section 5. For now, we briefly discuss these benefits, i.e., the rationale behind choosing one policy over the others.

The oldest-first policy aims to flush dirty pages which have been lingering the longest in the buffer pool. This is possible thanks to our CleanLSN mechanism introduced earlier. The goal of this policy is to reduce the log volume of the dirty page backlog, which in turn reduces the length of the REDO log scan during recovery. However, it does not necessarily decrease the number of dirty pages as much. For that, the coldest-first policy is more appropriate. It collects pages which are referenced the least, using a reference counter maintained in each frame descriptor. This reference counter can be reused by clock-based page replacement policies. The rationale behind flushing coldest pages first is that they are the less likely to become dirty again after flushing, and thus a significant reduction of the dirty page set is expected. Furthermore, they are the most likely to be selected for eviction, so cleaning them also improves the performance of the page replacement algorithm.

Finally, the policy of flushing hottest pages first may seem counter-intuitive, but it plays an important role for on-demand recovery schemes like instant restart [3]. In this case, it is worthwhile to reduce the recovery time of important pages such as system catalogs and B-tree roots. Since these tend to be the mostly accessed pages, this policy guarantees that they are always kept as up-to-date as possible. However, cold dirty pages will linger in the buffer pool without ever being flushed, and so the dirty page backlog is not reduced. Therefore, this policy is better utilized in combination with one of the other policies.

3.3 Problems of page-based propagation

The first problem of page-based propagation strategies is that they fail to sustain maximum write throughput. Despite the access pattern not being completely random, but jump-sequential thanks to the sorting of candidate frames, a large clustered page write is rare. Such large writes are required to deliver maximum throughput, especially in the case of synchronous writes.

A low write bandwidth alone is ineffective in reducing the dirty page backlog, but the fact that cleaning policies have such a complex parametrization space worsens the problem even further. Maximizing cleaner efficiency is a matter of choosing the ideal parameters for any point in time of a given workload. These parameters include not only the policy type, as discussed above, but also the number of candidates to choose at each iteration and whether to prioritize large clusters over single page writes—an additional dimension which was not considered in Algorithm 1.

Lastly, page-based propagation is tightly coupled to the buffer pool. As already observed by Levy and Silberschatz [7], propagation always causes some interference to normal transaction processing. The page cleaner loop presented in Algorithm 1 requires latching each flushed page three times: first to collect it as a candidate, then to copy it into the write buffer, and finally to update its CleanLSN. Furthermore, each dirty page not flushed must be accessed, and thus latched, at least once when collecting candidates. Despite being shared-mode latches, these may cause noticeable interference in a scenario of intensive transaction activity, which is also when page cleaning should run more aggressively.

4 Log-based propagation—a novel technique

Log-based propagation solves the aforementioned problems of page cleaning. First, its I/O pattern is purely sequential, which guarantees the best possible cleaning throughput. Second, because propagation is driven by the log, there is no need for any policy or prioritization scheme. Third, it does not interact with the buffer pool, thus reducing interference and increasing separation of concerns. This section introduces this new technique and elaborates on these advantages.

4.1 Partially sorted log

Log-based propagation could, in principle, rely on the transaction log to replay updates on the active database, but this would be inefficient given the random access pattern. This approach was proposed in related work [7], and the problem is recognized by the authors, who suggest an impractically large disk array to match the bandwidth of transaction updates.

A better approach is to reorganize the log so that log replay is performed sequentially. This idea was explored in previous work on single-pass restore [11], a technique to recover from media failures in a single sequential pass over log archive and backup. The technique consists of integrating a run generation phase in the archiving process, so that the log archive is composed of sorted runs. A run maps to a contiguous LSN range, but, within each run, log records are sorted primarily by page identifier. During restore, these runs are then merged to form a single sorted stream of log records. These two steps—run generation and merge—correspond to an external merge sort procedure, but because they are seamlessly integrated into normal processing and recovery, respectively, no noticeable overhead or increased downtime is incurred. We refer to the original publication for further details and experiments [11].

4.2 Log-based page cleaner

Similar to its page-based counterpart, the log-based cleaner runs in a dedicated thread. It runs Algorithm 2, presented here in pseudo-code, in a main loop. On each iteration, a subset of partitions in the partially sorted log is scanned, starting on the LSN on which the previous iteration stopped—here called $startLSN$. This delivers an iterator of log records sorted by page identifier (line 2).

Algorithm 2 Log-based cleaner main loop

```
 1: function LogBasedCleaner(sortedLog, startLSN)
 2:     iter ← sortedLog.open(startLSN)
 3:     buffer ← allocateBuffer()
 4:     while iter.hasNext() do
 5:         logrec ← iter.get()
 6:         readSegment(buffer, logrec.pid)
 7:         replayLog(buffer, iter)
 8:         flush(buffer)
 9:     end while
10:     return iter.endLSN
11: end function
```

The stream of sorted log records is processed one segment at a time, whereby a segment is defined as a fixed-size set of contiguous pages. This size should be such that scattered writes deliver good sequential write speed (e.g., 1 MB). Each segment is first read into the cleaner's internal buffer (line 6). Then, log replay is performed on this segment using the iterator, until the current log record refers to a page outside the current segment or the iterator has finished. At this point, the buffer is flushed into the persistent database and further segments are processed until the log scan iterator has finished. The end of the LSN range covered by the log scan is then returned to the caller—it will be used as the $startLSN$ on the next cleaner invocation.

Note that the algorithm has no reference to the buffer pool, which means that the page cleaner is completely decoupled from it. This has not only architectural advantages, i.e., better modularization and separation of concerns, but also performance benefits, since there is no latching or copying of pages in the buffer pool. We illustrate this in Figure 2. Traditional page-based propagation (on the left-hand side of the diagram) propagates data directly from in-memory data structures into persistent storage, creating a tight coupling between these components; unlike log-based propagation (on the right-hand side), where the components are independent. This decoupled design also has interesting properties that can be exploited in logging and recovery mechanisms—these are briefly discussed in Section 6. One detail worth mentioning is that the need for tracking dirty pages in the buffer pool is eliminated. However, this tracking is necessary if eviction of dirty pages is not allowed, i.e., if write elision [3] is not supported. To that end, an additional step is required in Algorithm 2 to mark pages flushed as clean. Because this design would introduce a dependency to the buffer pool module, it is not completely decoupled, but still fairly loosely coupled when compared with the traditional page cleaner.

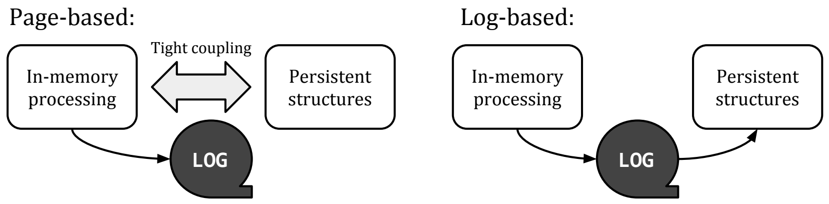

Fig. 2: Coupling of persistent and in-memory components

The log-based propagation algorithm has a jump-sequential I/O pattern, since segments are read and written in page-ID order, skipping segments for which no log record is found. If a moderately large segment size is used (e.g., a few megabytes for either SSD or HDD), this jump-sequential pattern fully utilizes the device sequential speed. One performance concern is that segments must be both read and written during propagation, which means that a single database device would spend roughly only half of the time performing writes. Furthermore, the log archive must also be read using a merge pattern, which may incur many random reads if too many log partitions are merged [11]. Thus, the I/O activity of the log-based cleaner is more intense than the traditional page-based approach. However, these problems are easily mitigated with simple software and hardware measures. First, if the partially sorted log is stored with redundancy (e.g., RAID-1)—which is a bare-minimal requirement for reliability—concurrent reads and writes can be performed in parallel. Second, if the merge logic of the log scan supports asynchronous read-ahead [2], then log reads are also performed in parallel with update propagation. Furthermore, despite this intense I/O behavior, the next section demonstrates that log-based propagation beats the traditional page cleaner even with a single non-redundant database device.

5 Experiments

5.1 Write bandwidth

Our first experiment analyzes the average write bandwidth sustained by 12 variations of page-based propagation strategies in comparison with the log-based strategy. For this experiment, which uses the TPC-C benchmark, the buffer pool is large enough to contain the whole dataset, which has initial size of 13 GB, and SSD devices are used for both log and database files. With 20 concurrent clients on a multi-core server, it delivers an average transaction throughput of 10,000 per second. Therefore, our goal here is to maximize pressure on the system and analyze how the propagation strategies keep up. The results are shown in Figure 3, with strategies on the x-axis and write bandwidth plotted in MB/s with a log scale on the y-axis.

The first nine strategies consist of the three policies described in Section 3.2 using three different sizes for the priority queue of candidate frames—2,000, 20,000 and 200,000. This corresponds roughly to 0.1%, 1%, and 10% of the application working set, respectively. We note that all of them are quite slow, utilizing only from 3 to 5 MB/s write bandwidth. This is because most writes are of single pages, which is inefficient even for SSD devices. Three other page-based strategies are considered in this experiment. The first one, labeled "no-policy" is a naive strategy in which every dirty frame is flushed, thus ignoring any prioritization policy. At ∼6 MB/s, it is slightly more efficient than the others, because more opportunities for large writes are found. The two "clustered" policies are just like "no-policy", but only page writes larger than a certain number of pages

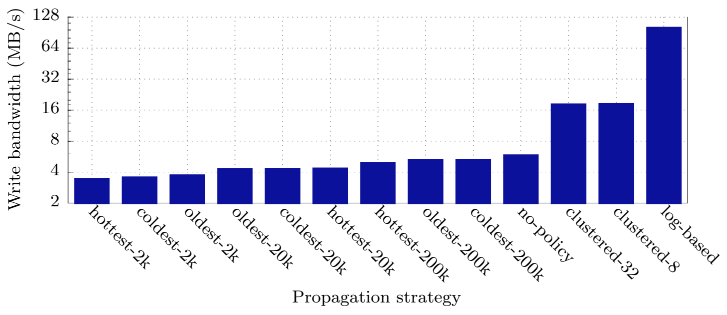

Fig. 3: Write bandwidth of different propagation strategies

are performed—here 8 and 32 pages. We note that the bandwidth is indeed increased to about 18 MB/s, but the policy is not as effective because the larger the minimum size is, the less likely it is that large-enough clusters are found; this is why the 8-page policy is slightly faster than the 32-page policy. Finally, the log-based propagation strategy, which has only a single variant, is by far the most efficient, at 100 MB/s. The maximum bandwidth of the device is actually 200 MB/s, but, as discussed earlier, half of the time is spend performing reads, which means that 100 MB/s is indeed the maximum possible speed for this propagation strategy.

5.2 Backlog reduction

The next experiment analyzes the effectiveness of propagation strategies in reducing the dirty page backlog. We break down the execution of each experiment into a time series of 20 minutes and measure the number of dirty pages as well as the log volume covered by them, i.e., the length of the REDO log scan in case of a system failure. Because the page-based policies introduced in Section 3.2 are very inefficient with a single storage device, we consider—in addition to the scenario of the previous experiment—a low-throughput scenario with ∼1,000 transactions per second. With the lower dirtying speed, page-base strategies should be more effective and interesting comparisons may be drawn.

The results are shown in Figure 4. The two plots on the top correspond to the low-throughput scenario, whereas the bottom plots are high-throughput ones. The plots on the left-hand side measure the number of dirty pages in the buffer pool, while the ones on the right-hand side measure the REDO length. For this experiment, we consider only three page-based policies: oldest-first with 200,000 candidate frames; the clustered strategy with 8 pages; and a "mixed" policy which is a special version of oldest-first—it ignores newly allocated, never-flushed pages in 3/4 of the cleaner activations. We implemented this strategy to show that mixing policies and adjusting parameters allows for more efficient cleaning when tailored to a particular workload. Other strategy variants have similar results and thus provide no additional insight.

For the low-throughput scenario, we observe that the clustered policy, as expected, is not able to reduce the dirty page backlog despite delivering better write bandwidth. The oldest-first policy is able to maintain a low dirty page count between 20,000 and 30,000, but it performs just as bad as the clustered strategy in controlling REDO length. Our mixed strategy tailored for this workload actually performs best on both criteria: it maintains a stable and low dirty page count (after an initial period of instability) and is more effective than the two other page-based policies in controlling REDO length. The log-based propagation strategy maintains a higher dirty page count than the mixed and oldest-first policies—this can be attributed to the natural backlog occurring due to the delay between inserting a log record in the (unsorted) recovery log and processing it in the partially sorted log. The zig-zag pattern is a consequence of the log-based cleaning algorithm, which processes runs of the partially sorted log and segments of multiple pages in batches. It performs slightly better than the mixed policy in

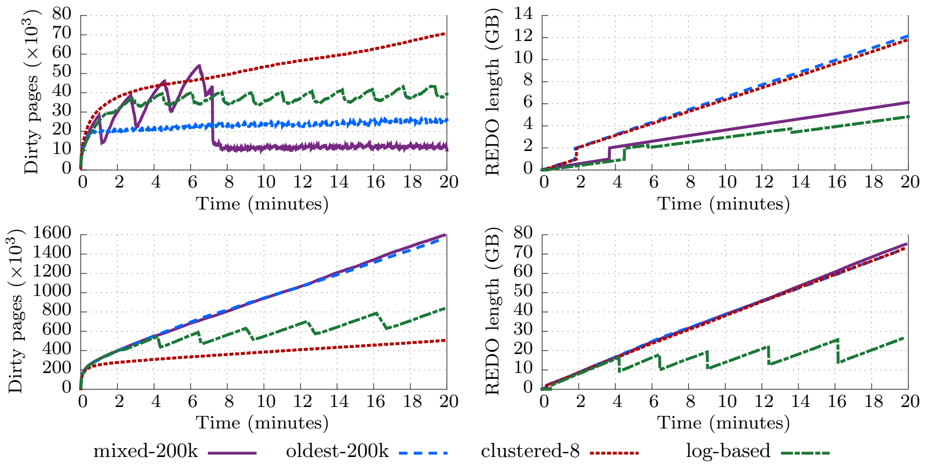

Fig. 4: Backlog analysis for low- (top) and high-throughput (bottom) scenarios

REDO length, but the main take-away here is that none of the strategies is able to maintain it stable, suggesting that an adaptive approach, possibly combining both log- and page-based propagation, might be more appropriate.

In the high-throughput scenario, which is the main goal of our investigation, log-based propagation performs better than the mixed and oldest-first policies, but loses to the clustered approach in maintaining a low dirty page count. However, it is the only approach which is able to control the REDO length, with a large margin to page-based strategies. Thus, these results clearly demonstrate its superiority for the workload considered.

6 Outlook and conclusion

This work deals with the problem of update propagation for high-performance OLTP scenarios. Given the ever-increasing performance gap between in-memory processing and I/O operations, as well as the decreasing costs of main memory, a database system's buffer pool may get saturated with dirty data, unless an efficient propagation strategy is employed. This makes it more challenging to maintain a well-balanced system using hardware alone. The approaches presented here address the problem with software techniques, improving hardware utilization and thus reducing costs.

We described a flexible page-based propagation tool (the page cleaner) and analyzed its effectiveness under a variety of policies. Our empirical evaluation shows that this traditional approach is not able to fully exploit the write bandwidth of a single storage device. In addition to the inefficiency problem, we pointed out the tight coupling between buffer management and persistence modules in the traditional design. The storage manager of a database system is known in the literature for having intricate dependencies between its components: concurrency control, recovery, buffer management, and storage structures [6]. This is not only an architectural problem for code maintenance, reusability, and evolution, but also a performance problem for scalability of transactional workloads.

To solve these two problems—cleaning inefficiency and tight coupling—we proposed a log-based propagation strategy. Instead of flushing dirty pages from the buffer pool directly into persistent storage, an independent system component propagates updates into the persistent database using log replay. To support a sequential access pattern, a partially sorted log data structure is borrowed from previous work in the context of recovery from media failures [11]. Our empirical evaluation shows that log-based propagation is able to fully utilize the bandwidth of the database device, thus providing much higher cleaner efficiency. In practice, this results in reduced operational costs, as less disks are required to match in-memory performance and reach a balanced state. Lastly, this novel propagation technique does not require any access to the buffer pool data structures, simplifying the buffer manager implementation and increasing separation of concerns in the system architecture.

References

1. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M., Wood, D.A.: Implementation techniques for main memory database systems. In: Proc. SIGMOD. pp. 1–8 (1984)
2. Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2), 73–170 (1993)
3. Graefe, G., Guy, W., Sauer, C.: Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2014)
4. Graefe, G., Volos, H., Kimura, H., Kuno, H.A., Tucek, J., Lillibridge, M., Veitch, A.C.: In-memory performance for big data. PVLDB 8(1), 37–48 (2014)
5. Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Computing Surveys 15(4), 287–317 (1983)
6. Hellerstein, J.M., Stonebraker, M., Hamilton, J.: Architecture of a database system. Now Publishers Inc. (2007)
7. Levy, E., Silberschatz, A.: Log-driven backups: A recovery scheme for large memory database systems. In: Proc. 5th Jerusalem Conference on Information Technology. pp. 99–109 (1990)
8. Malviya, N., Weisberg, A., Madden, S., Stonebraker, M.: Rethinking main memory OLTP recovery. In: Proc. ICDE. pp. 604–615 (2014)
9. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17(1), 94–162 (1992)
10. Sauer, C., Graefe, G., Härder, T.: An empirical analysis of database recovery costs. In: RDSS (SIGMOD Workshops), Snowbird, UT, USA (2014)
11. Sauer, C., Graefe, G., Härder, T.: Single-pass restore after a media failure. In: Proc. BTW, LNI 241. pp. 217–236 (2015)